# pylint: disable=no-member, arguments-differ, attribute-defined-outside-init, unused-argument

"""

Implementing Full Sparse Layer

"""

import math

from twitter.deepbird.sparse import sparse\_dense\_matmul

from .layer import Layer

import tensorflow.compat.v1 as tf

import twml

class FullSparse(Layer):

"""Fully-sparse layer class.

This layer implements the operation:

.. code-block:: python

outputs = activation(inputs.weight + bias)

Arguments:

output\_size:

Long or Integer, dimensionality of the output space.

input\_size:

The number of input units. (Deprecated)

weight\_initializer:

Initializer function for the weight matrix.

This argument defaults to zeros\_initializer().

This is valid when the FullSparse is the first layer of

parameters but should be changed otherwise.

weight\_regularizer:

Regularizer function for the weight matrix.

Ensure to add tf.losses.get\_regularization\_loss() to your loss for this to take effect.

bias\_regularizer:

Regularizer function for the bias.

Ensure to add tf.losses.get\_regularization\_loss() to your loss for this to take effect

activation:

Activation function (callable). Set it to None to maintain a linear activation.

bias\_initializer:

Initializer function for the bias.

This argument defaults to tf.constant\_initializer(1/output\_size)

trainable:

Boolean, if `True` also add variables to the graph collection

``GraphKeys.TRAINABLE\_VARIABLES`` (see `tf.Variable

<https://www.tensorflow.org/versions/master/api\_docs/python/tf/Variable>`\_).

name:

String, the name of the layer. Layers with the same name will

share weights, but to avoid mistakes we require ``reuse=True`` in such cases.

use\_sparse\_grads:

Boolean, if `True` do sparse mat mul with `embedding\_lookup\_sparse`, which will

make gradients to weight matrix also sparse in backward pass. This can lead to non-trivial

speed up at training time when input\_size is large and optimizer handles sparse gradients

correctly (eg. with SGD or LazyAdamOptimizer). If weight matrix is small, it's recommended

to set this flag to `False`; for most use cases of FullSparse, however, weight matrix will

be large, so it's better to set it to `True`

num\_partitions:

Number of partitions to use for the weight variable. Defaults to 1.

partition\_axis:

If num\_partitions is specified, the partition axis for the weight variable

Defaults to 0 (partition by row).

Must be 0 (row) or 1 (column)

use\_binary\_values:

Assume all non zero values are 1. Defaults to False.

This can improve training if used in conjunction with MDL.

This parameter can also be a list of binary values if `inputs` passed to `call` a list.

use\_compression:

Default False. Set True to enable data compression techniques for

optimization of network traffic for distributed training.

use\_binary\_sparse\_dense\_matmul:

If binary sparse dense matmul op is to be used. It will only be enabled if

`use\_binary\_values` is set true. It only should be used for inference, best practice is

to set `use\_binary\_sparse\_dense\_matmul = not is\_training`.

"""

def \_\_init\_\_(self,

output\_size,

input\_size=None,

weight\_initializer=None,

activation=None,

bias\_initializer=None,

trainable=True,

name=None,

use\_sparse\_grads=True,

num\_partitions=None,

partition\_axis=0,

use\_binary\_values=False,

bias\_regularizer=None,

weight\_regularizer=None,

use\_compression=False,

use\_binary\_sparse\_dense\_matmul=False,

\*\*kwargs):

super(FullSparse, self).\_\_init\_\_(trainable=trainable, name=name, \*\*kwargs)

# TODO - remove input\_size warning.

if input\_size:

raise ValueError('input\_size is deprecated - it is now automatically \

inferred from your input.')

# The bias initialization and weights initialization is set to match v1's implementation.

if bias\_initializer is None:

bias\_initializer = tf.constant\_initializer(1 / output\_size)

# Weights initialization is set to 0s. This is safe for full sparse layers because

# you are supposed to learn your embedding from the label.

if weight\_initializer is None:

weight\_initializer = tf.zeros\_initializer()

self.weight\_initializer = weight\_initializer

self.bias\_initializer = bias\_initializer

self.output\_size = output\_size

self.activation = activation

self.use\_sparse\_grads = use\_sparse\_grads

self.num\_partitions = num\_partitions

if partition\_axis != 0 and partition\_axis != 1:

raise ValueError('partition\_axis must be 0 or 1')

self.partition\_axis = partition\_axis

self.use\_binary\_values = use\_binary\_values

self.weight\_regularizer = weight\_regularizer

self.bias\_regularizer = bias\_regularizer

self.\_use\_compression = use\_compression

self.\_cast\_indices\_dtype = tf.int32 if self.\_use\_compression else None

self.use\_binary\_sparse\_dense\_matmul = use\_binary\_sparse\_dense\_matmul

def \_make\_weight\_var(self, shape, partitioner):

self.weight = self.add\_variable(

'weight',

initializer=self.weight\_initializer,

regularizer=self.weight\_regularizer,

shape=shape,

dtype=self.dtype,

trainable=True,

partitioner=partitioner,

)

def build(self, input\_shapes):

"""

creates the ``bias`` and ``weight`` Variables

of shape ``[output\_size]`` and ``[input\_size, output\_size]`` respectively.

"""

if isinstance(input\_shapes, (list, tuple)):

input\_shape = input\_shapes[0]

is\_compatible = True

for other\_shape in input\_shapes[1:]:

is\_compatible &= input\_shape.is\_compatible\_with(other\_shape)

if not is\_compatible:

raise ValueError("Input shapes %s are not compatible." % input\_shapes)

else:

input\_shape = input\_shapes

self.bias = self.add\_variable(

'bias',

initializer=self.bias\_initializer,

regularizer=self.bias\_regularizer,

shape=[self.output\_size, ],

dtype=self.dtype,

trainable=True

)

partitioner = None

shape = [input\_shape[1], self.output\_size]

# There is a 2gb limitation for each tensor because of protobuf.

# 2\*\*30 is 1GB. 2 \* (2\*\*30) is 2GB.

dtype = tf.as\_dtype(self.dtype)

num\_partitions = 1 if self.num\_partitions is None else self.num\_partitions

in\_shape = input\_shape[1]

out\_shape = self.output\_size

# when v2 behavior is disabled, in\_shape is tf.Dimension. otherwise it is int.

if isinstance(in\_shape, tf.Dimension):

in\_shape = in\_shape.value

if in\_shape is None:

raise ValueError("Input tensor should have shape."

" You can set it using twml.util.limit\_sparse\_tensor\_size")

(split\_dim, other\_dim) = (in\_shape, out\_shape) if self.partition\_axis == 0 else (out\_shape, in\_shape)

requested\_size = math.ceil(float(split\_dim) / num\_partitions) \* other\_dim \* dtype.size

if (requested\_size >= 2\*\*31):

raise ValueError("Weight tensor partitions cannot be larger than 2GB.\n"

"Requested Dimensions(%d, %d) of type %s (%d bytes total) over %d partitions.\n"

"Possible solutions:\n"

"- reduce the params.output\_size\_bits\n"

"- reduce the output\_size of the sparse\_layer\n"

"- specify a larger num\_partitions argument\n"

"- reduce input\_size\_bits" %

(in\_shape, self.output\_size, dtype.name, requested\_size, num\_partitions))

if self.num\_partitions:

partition\_axis = int(self.partition\_axis)

partitioner = tf.fixed\_size\_partitioner(self.num\_partitions, axis=partition\_axis)

else:

# Regular variables do not like it when you pass both constant tensors and shape

if not callable(self.weight\_initializer):

shape = None

self.\_make\_weight\_var(shape, partitioner)

self.built = True

def compute\_output\_shape(self, input\_shape):

"""Computes the output shape of the layer given the input shape.

Args:

input\_shape: A (possibly nested tuple of) `TensorShape`. It need not

be fully defined (e.g. the batch size may be unknown).

Raises NotImplementedError.

"""

raise NotImplementedError

def call(self, inputs, \*\*kwargs): # pylint: disable=unused-argument

"""The logic of the layer lives here.

Arguments:

inputs:

A SparseTensor or a list of SparseTensors.

If `inputs` is a list, all tensors must have same `dense\_shape`.

Returns:

- If `inputs` is `SparseTensor`, then returns `bias + inputs \* dense\_b`.

- If `inputs` is a `list[SparseTensor`, then returns

`bias + add\_n([sp\_a \* dense\_b for sp\_a in inputs])`.

"""

if isinstance(inputs, (list, tuple)):

if isinstance(self.use\_binary\_values, (list, tuple)):

use\_binary\_values = self.use\_binary\_values

else:

use\_binary\_values = [self.use\_binary\_values] \* len(inputs)

num\_inputs = len(inputs)

if num\_inputs != len(use\_binary\_values):

raise ValueError("#inputs is %d while #use\_binary\_values is %d"

% (num\_inputs, len(use\_binary\_values)))

outputs = []

for n in range(num\_inputs):

outputs.append(sparse\_dense\_matmul(inputs[n], self.weight,

self.use\_sparse\_grads,

use\_binary\_values[n],

name='sparse\_mm\_' + str(n),

partition\_axis=self.partition\_axis,

num\_partitions=self.num\_partitions,

compress\_ids=self.\_use\_compression,

cast\_indices\_dtype=self.\_cast\_indices\_dtype,

use\_binary\_sparse\_dense\_matmul=self.use\_binary\_sparse\_dense\_matmul))

outputs = tf.accumulate\_n(outputs)

else:

if isinstance(self.use\_binary\_values, (list, tuple)):

raise ValueError("use\_binary\_values can not be %s when inputs is %s" %

(type(self.use\_binary\_values), type(inputs)))

outputs = sparse\_dense\_matmul(inputs, self.weight,

self.use\_sparse\_grads,

self.use\_binary\_values,

name='sparse\_mm',

partition\_axis=self.partition\_axis,

num\_partitions=self.num\_partitions,

compress\_ids=self.\_use\_compression,

cast\_indices\_dtype=self.\_cast\_indices\_dtype,

use\_binary\_sparse\_dense\_matmul=self.use\_binary\_sparse\_dense\_matmul)

if self.bias is not None:

outputs = tf.nn.bias\_add(outputs, self.bias)

if self.activation is not None:

return self.activation(outputs) # pylint: disable=not-callable

return outputs

def full\_sparse(

inputs, output\_size,

input\_size=None,

activation=None,

bias\_regularizer=None,

weight\_regularizer=None,

bias\_initializer=None,

weight\_initializer=None,

trainable=True,

name=None,

reuse=None,

use\_sparse\_grads=True,

num\_partitions=None,

partition\_axis=0,

use\_binary\_values=False,

use\_compression=False):

"""Functional interface for the sparsely-connected layer.

Arguments:

inputs:

A sparse tensor (can be twml.SparseTensor or tf.SparseTensor)

output\_size:

Long or Integer, dimensionality of the output space.

weight\_initializer:

Initializer function for the weight matrix.

activation:

Activation function (callable). Set it to None to maintain a linear activation.

bias\_initializer:

Initializer function for the bias.

weight\_regularizer:

Regularizer function for the weight matrix.

Ensure to add tf.losses.get\_regularization\_loss() to your loss for this to take effect.

bias\_regularizer:

Regularizer function for the bias.

Ensure to add tf.losses.get\_regularization\_loss() to your loss for this to take effect.

trainable:

Boolean, if `True` also add variables to the graph collection

``GraphKeys.TRAINABLE\_VARIABLES`` (see `tf.Variable

<https://www.tensorflow.org/versions/master/api\_docs/python/tf/Variable>`\_).

name:

String, the name of the layer. Layers with the same name will

share weights, but to avoid mistakes we require ``reuse=True`` in such cases.

use\_sparse\_grads:

Boolean, if `True` do sparse mat mul with `embedding\_lookup\_sparse`, which will

make gradients to weight matrix also sparse in backward pass. This can lead to non-trivial

speed up at training time when input\_size is large and optimizer handles sparse gradients

correctly (eg. with SGD or LazyAdamOptimizer). If weight matrix is small, it's recommended

to set this flag to `False`; for most use cases of FullSparse, however, weight matrix will

be large, so it's better to set it to `True`

num\_partitions:

Number of partitions to use for the weight variable. Defaults to 1.

partition\_axis:

If num\_partitions is specified, the partition axis for the weight variable

Defaults to 0 (partition by row).

Must be 0 (row) or 1 (column)

use\_binary\_values:

Assume all non zero values are 1. Defaults to False.

This can improve training if used in conjunction with MDL.

use\_compression:

Default False. Set True to enable data compression techniques for

optimization of network traffic for distributed training.

Returns:

Outputs a ``tf.Tensor`` of size ``[batch\_size x output\_size]``.

"""

# TODO - remove input\_size warning.

if input\_size:

raise ValueError('input\_size is deprecated - it is now \

automatically inferred from your input.')

dtype = None

if isinstance(inputs, twml.SparseTensor):

inputs = inputs.to\_tf()

dtype = inputs.dtype.base\_dtype

if isinstance(inputs, (list, tuple)):

inputs = [inp.to\_tf() if isinstance(inp, twml.SparseTensor) else inp for inp in inputs]

dtype = inputs[0].dtype.base\_dtype

layer = FullSparse(output\_size=output\_size,

activation=activation,

trainable=trainable,

name=name,

weight\_initializer=weight\_initializer,

bias\_initializer=bias\_initializer,

weight\_regularizer=weight\_regularizer,

bias\_regularizer=bias\_regularizer,

dtype=dtype,

\_scope=name,

\_reuse=reuse,

use\_sparse\_grads=use\_sparse\_grads,

num\_partitions=num\_partitions,

partition\_axis=partition\_axis,

use\_compression=use\_compression,

use\_binary\_values=use\_binary\_values)

return layer(inputs)